Naïve Version

• Warning: this code is simplified/incorrect!

```java
public class Map<K, V> {
    protected V[] entries;

    public V get(K key) {
        int index = key.hashCode() % entries.length;
        return entries[index];
    }

    public void put(K key, V value) {
        int index = key.hashCode() % entries.length;
        entries[index] = value;
    }
}
```
Hash Collisions

- `k1.hashCode() == k2.hashCode() but k1 != k2`
  - May also be caused by the modulus operation
- This is inevitable (e.g., the birthday paradox)
- A “good” hash function rarely collides
Two main strategies to avoid collisions

1) Open addressing (closed hashing):
   Each bucket can store at most one entry
   If hash falls in occupied bucket then search procedure for
   next empty bucket based on:
   - Linear probing
   - Quadratic probing
   - Double probing

2) Closed Addressing (open or external hashing/bucketing):
   Each bucket can store multiple entries
   - Separate chaining
Linear Probing

• If we collide, check next entry until one is empty. Wrap around when at the end of table

• Deletion is complicated

• Can only hold entries.length items

• Resizing the table requires rehashing everything

• Suffers from primary clustering
### Linear Probing Example: $h(k) = k \% 13$

Keys to insert: 17, 33, 18, 20, 44, 11, 19, 7 (ignore values)

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td>18</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td>18</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td>18</td>
<td>33</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td>18</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td>18</td>
<td>33</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td>18</td>
<td>33</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td>18</td>
<td>33</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td>18</td>
<td>33</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td>18</td>
<td>33</td>
</tr>
</tbody>
</table>

Collision!
Linear Probing

• Keys with same hash will be clustered together
• The same thing can happen with unrelated keys forming primary clusters
• The more elements we add, the more collisions
Linear Probing Lookup

• Start at location returned by hashing function
  • If key was found → value
  • If key was not found search linearly until:
    • You find the key → value
    • You find an empty slot before you have found key → null
    • You wrapped around and ended up where you started → null

• Example: get(7) returns the value for 7

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>17</td>
<td>18</td>
<td>44</td>
<td>33</td>
<td>20</td>
<td>19</td>
<td>7</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

• Example: get(6) returns null

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>17</td>
<td>18</td>
<td>44</td>
<td>33</td>
<td>20</td>
<td>19</td>
<td>7</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>
Quadratic Probing

- \( h(k, i) = (h(k) + c_1 i + c_2 i^2) \mod n \), \( c_2 \neq 0 \)
- If \( c_2 = 0 \) then degrades to linear probing
- E.g., \( h(k, i) = (h(k) + i^2) \mod n \), then for every probing \( h(k), h(k) + 1, h(k) + 4, \ldots \)
- Can result in cases where we don’t try all slots
  - E.g., \( n = 5 \), and start with \( h(k) = 1 \).
  - Rehashings give 2, 0, 0, 2, 1, 2, 0, 0, ...
  - The slots 3 and 4 will never be examined to see if they have room
- Secondary Clustering
Quadratic Probing: $h(k, i) = (k\%13) + i^2$

Keys to insert: 17, 33, 18, 20, 44, 11, 19, 7 (ignore values)

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

17

17

17 33

17 18 33

17 18 33 20

17 18 44 33 20

17 18 44 33 20 11

17 18 44 33 20 19 11

7 17 18 44 33 20 19 11

Collision!

Collision!

Collision!

Collision!

Collision!
Double Hashing

• Use second hash function on key to determine delta (interval) for next try

• $h(k, i) = (h_1(k) + i \cdot h_2(k)) \pmod{n}$,

• E.g., $h_2(k) = (k \mod (n - 2)) + 1$

• Helps with primary and secondary clustering

• Example:
  • Suppose $h_1(n) = n \% 5$
  • Then $h_1(1) = h_1(6) = h_1(11)$
  • However, $h_2(1) = 2, h_2(6) = 1, h_2(11) = 3$
Separate Chaining

• Turn each bucket into a linked list (or array, etc.)

• On collision add to the bucket

• Searching list is fast if lists are small

• Deletion is simple

• Can hold more than entries.length items easily
Load Factor

- Performance depends on *load factor*
- Load factor is \( \alpha = \frac{n}{N} \) where \( n \) = items in table and \( N \) = size of table
- Higher load factor \( \rightarrow \) more collisions \( \rightarrow \) slow
- Can be > 1 for external chaining
- For open addressing usually want to ensure \( \alpha < 0.75 \)
  - Generally \( \alpha > 0.75 \) means resize the table (& rehash everything)
## Performance

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Unsuccessful</th>
<th>Successful</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Probing</td>
<td>$\frac{1}{2} \left(1 + \frac{1}{(1 - a)^2}\right)$</td>
<td>$\frac{1}{2} \left(1 + \frac{1}{(1 - a)}\right)$</td>
</tr>
<tr>
<td>Double Probing</td>
<td>$\frac{1}{1 - a}$</td>
<td>$-\frac{1}{a} \log(1 - a)$</td>
</tr>
<tr>
<td>External Chaining</td>
<td>$a + e^{-a}$</td>
<td>$1 + a/2$</td>
</tr>
</tbody>
</table>

Entries represent expected number of comparisons needed to find a specific element (successful) or demonstrate that it is not in the hash table (unsuccessful).

$a$ is the load factor.
Performance for $a = .9$

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Unsuccessful</th>
<th>Successful</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Probing</td>
<td>55</td>
<td>5.5</td>
</tr>
<tr>
<td>Double Probing</td>
<td>10</td>
<td>~4</td>
</tr>
<tr>
<td>External Chaining</td>
<td>3</td>
<td>1.45</td>
</tr>
</tbody>
</table>

Entries represent number of comparisons needed to find a specific element or demonstrate that it is not in the hash table
Space requirements

• Open addressing: TableSize + n*objectsize
• External chaining: TableSize + n*(objectsize+1)

• Rule of thumb:
  • Small elements, small load factor: open addressing
  • Large elements, large load factor: external chaining